Project report : Risk of fire prediction

Idriss EL WAANABI

July 2, 2024

Abstract

The increasing frequency and severity of wildfires pose a significant threat to both natural ecosystems and human settlements in Morocco. This project aims to develop a robust predictive model to assess fire risk across various regions in Morocco. By leveraging historical fire data, meteorological information, vegetation indices, and socio-economic factors, we intend to identify key determinants of fire outbreaks and predict potential fire-prone areas.

The methodology involves data collection, preprocessing, and the application of advanced machine learning algorithms. Key steps include feature selection, model training and validation using techniques such as Random Forest, Gradient Boosting, and Neural Networks, and continuous model refinement to enhance predictive accuracy.

1. Data collection and exploration

The central component of this dataset is a comprehensive file named Date final dataset balanced float32.parquet, formatted in Parquet for efficiency and optimized for machine learning workflows. This meticulously balanced dataset comprises 934,586 rows and 278 columns, encapsulating a diverse range of features essential for accurate wildfire prediction models.

Detailed Description of `Date_final_dataset_balanced_float32.parquet`:

- **Format:** Parquet (optimized for efficient data storage and retrieval, with float32 precision to balance detail and storage requirements).
- **Shape:** 934,586 rows $\times$ 278 columns, providing a robust dataset for ML modeling.
- **Balance:** The dataset is balanced with equal instances of wildfire and non-wildfire occurrences to aid in unbiased model training.
- **Features Include:**
  - Temporal markers like `acq_date`, `day_of_week`, `day_of_year`, and holiday-related flags (`is_holiday`, `is_weekend`).
  - Spatial features such as latitude, longitude, and sea distance.

– Environmental conditions, including NDVI for vegetation health and soil moisture.

– Weather data lag features for up to 15 days prior, covering `average_temperature`, `maximum_temperature`, `minimum_temperature`, precipitation, snow depth, wind speed, and more, providing a comprehensive view of conditions leading up to an event.

– Aggregated weather statistics and historical averages over weekly, monthly, quarterly, and yearly periods to capture long-term trends and seasonal impacts on wildfire occurrences.

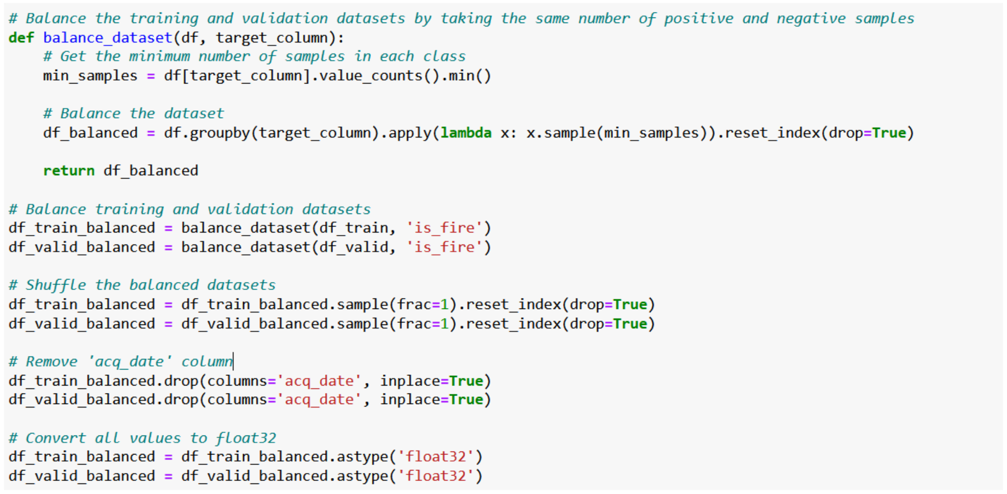
– The outcome variable `is_fire` indicates the presence (1) or absence (0) of a wildfire, serving as the target for predictive modeling.

This file's extensive feature set, including both raw and engineered attributes, is designed to support advanced machine learning techniques for predicting wildfire risks. By incorporating a broad spectrum of temporal, spatial, environmental, and meteorological data, it enables the creation of nuanced models that can accurately predict wildfire occurrences in Morocco.

The lines of code `df_train = df[df.acq_date < '2022-01-01']` and `df_valid = df[df.acq_date >= '2022-01-01']` are used to split a DataFrame `df` into training and validation datasets based on the acquisition date (`acq_date`). The training dataset `df_train` includes all rows with dates before January 1, 2022, while the validation dataset `df_valid` includes rows with dates from January 1, 2022, onwards. This temporal split ensures that the model is trained on past data and validated on more recent data, mimicking a real-world scenario where future events are predicted based on historical data, thus preventing data leakage and providing a realistic evaluation of the model's performance.

- Balancing dataset

  The given code snippet balances the training and validation datasets by ensuring that

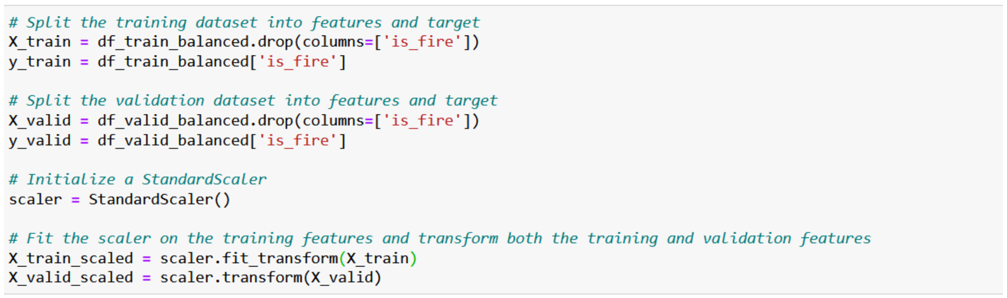


```python
# Balance the training and validation datasets by taking the same number of positive and negative samples
def balance_dataset(df, target_column):
    # Get the minimum number of samples in each class
    min_samples = df[target_column].value_counts().min()

    # Balance the dataset
    df_balanced = df.groupby(target_column).apply(lambda x: x.sample(min_samples)).reset_index(drop=True)

    return df_balanced

# Balance training and validation datasets
df_train_balanced = balance_dataset(df_train, 'is_fire')
df_valid_balanced = balance_dataset(df_valid, 'is_fire')

# Shuffle the balanced datasets
df_train_balanced = df_train_balanced.sample(frac=1).reset_index(drop=True)
df_valid_balanced = df_valid_balanced.sample(frac=1).reset_index(drop=True)

# Remove 'acq_date' column
df_train_balanced.drop(columns='acq_date', inplace=True)
df_valid_balanced.drop(columns='acq_date', inplace=True)

# Convert all values to float32
df_train_balanced = df_train_balanced.astype('float32')
df_valid_balanced = df_valid_balanced.astype('float32')
```

there are equal numbers of positive and negative samples. This is achieved through a function that first determines the minimum number of samples in each class within the target column. It then samples this minimum number from each class, resulting in a balanced dataset.

The function is applied to both the training and validation datasets, producing balanced versions of each. These balanced datasets are then shuffled to ensure randomness. Afterward, the `acq_date` column is removed, as it is not needed for model training. Finally, all values in the balanced datasets are converted to the `float32` data type to optimize memory usage and computational efficiency.

- Spliting data
  The training and validation datasets are prepared by first separating them into fea-



```
# Split the training dataset into features and target
X_train = df_train_balanced.drop(columns=['is_fire'])
y_train = df_train_balanced['is_fire']

# Split the validation dataset into features and target
X_valid = df_valid_balanced.drop(columns=['is_fire'])
y_valid = df_valid_balanced['is_fire']

# Initialize a StandardScaler
scaler = StandardScaler()

# Fit the scaler on the training features and transform both the training and validation features
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)
```

  tures and target variables. The features (`X_train` and `X_valid`) are extracted from their respective datasets (`df_train_balanced` and `df_valid_balanced`) by excluding the target variable 'is_fire', while the target variables (`y_train` and `y_valid`) specifically capture whether each instance represents a fire ('is_fire'). Subsequently, a `StandardScaler` from scikit-learn is employed to standardize the features. It first fits the scaler on the training features (`X_train`), computing the mean and standard deviation used for scaling, and then transforms both the training (`X_train_scaled`) and validation features (`X_valid_scaled`) to a standardized scale using these parameters.

2. Machine learning approach

Logistic Regression:

Logistic regression is a linear model used for binary classification tasks, such as predicting whether an instance belongs to one of two classes (in this case, fire or non-fire). Here's how it works:

- **Model Representation:** Logistic regression models the probability that an instance belongs to a particular class using the logistic function, which maps any real-valued input to a value between 0 and 1.
- **Decision Boundary:** It computes a decision boundary based on the feature values. If the estimated probability (output of the logistic function) is greater than 0.5, it predicts the instance belongs to the positive class (fire); otherwise, it predicts the negative class (non-fire).

- **Training:** During training, logistic regression learns the optimal coefficients (weights) for each feature through optimization techniques like gradient descent, aiming to minimize the error between predicted probabilities and actual class labels.

Random Forest:

Random forest is an ensemble learning method that operates by constructing multiple decision trees during training and outputs the mode of the classes (classification) or the average prediction (regression) of the individual trees.

- **Ensemble of Decision Trees:** It builds a forest of decision trees, where each tree is trained on a random subset of the training data (bootstrap sampling) and a random subset of the features.
- **Decision Making:** During prediction, each tree in the forest independently predicts the class, and the final prediction is determined by a majority vote (for classification) or averaging (for regression) across all trees.
- **Benefits:** Random forests are robust against overfitting, handle high-dimensional datasets well, and can capture non-linear relationships between features and the target variable.

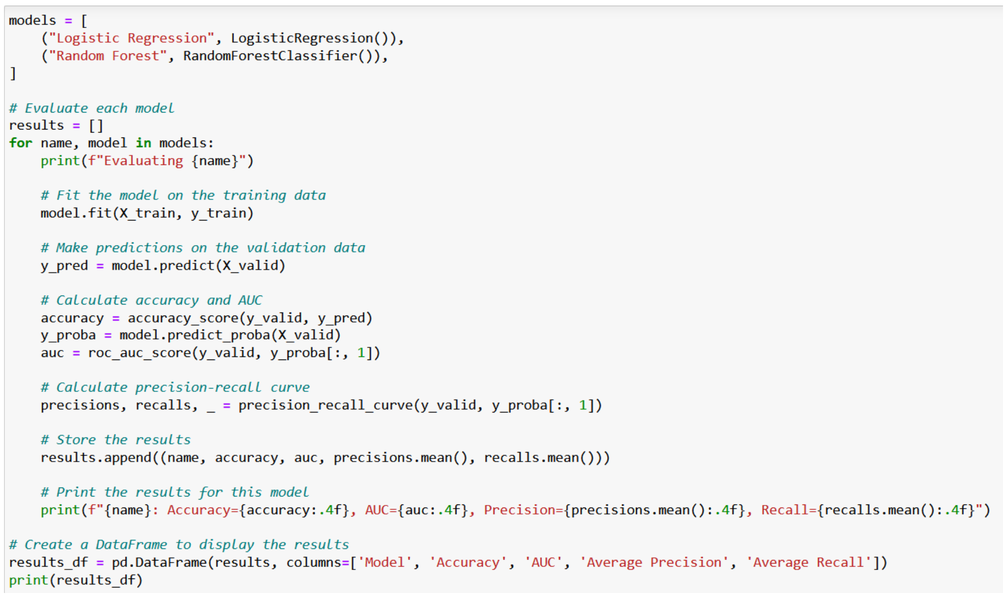
Application to the Case:

- **Logistic Regression:** Suitable when there is a linear relationship between features and the log-odds of the target variable (probability of fire in this case). It is computationally efficient and provides interpretable results in terms of feature importance.
- **Random Forest:** Useful when the relationship between features and the target variable is non-linear or when interactions between features are important. It generally provides higher accuracy than logistic regression but may be harder to interpret due to the ensemble nature.

Considerations:

- **Data Size:** Logistic regression can handle large datasets efficiently. Random forests can also handle large datasets but may be slower to train and predict compared to logistic regression.
- **Interpretability:** Logistic regression provides coefficients that indicate the impact of each feature on the probability of fire. Random forests offer feature importance scores but may not provide direct insights into the relationships between features and target.

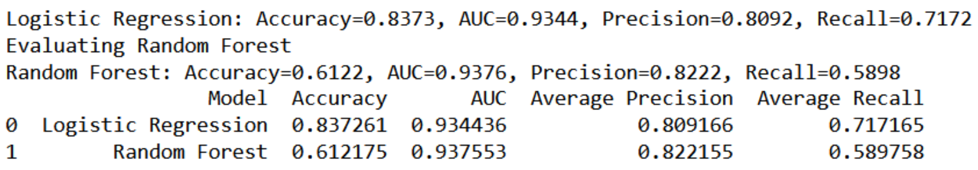
This code evaluates two machine learning models, Logistic Regression and Random Forest, on a dataset divided into training (`X_train`, `y_train`) and validation (`X_valid`,



```python
models = [
    ("Logistic Regression", LogisticRegression()),
    ("Random Forest", RandomForestClassifier()),
]

# Evaluate each model
results = []
for name, model in models:
    print(f"Evaluating {name}")

    # Fit the model on the training data
    model.fit(X_train, y_train)

    # Make predictions on the validation data
    y_pred = model.predict(X_valid)

    # Calculate accuracy and AUC
    accuracy = accuracy_score(y_valid, y_pred)
    y_proba = model.predict_proba(X_valid)
    auc = roc_auc_score(y_valid, y_proba[:, 1])

    # Calculate precision-recall curve
    precisions, recalls, _ = precision_recall_curve(y_valid, y_proba[:, 1])

    # Store the results
    results.append((name, accuracy, auc, precisions.mean(), recalls.mean()))

    # Print the results for this model
    print(f"{name}: Accuracy={accuracy:.4f}, AUC={auc:.4f}, Precision={precisions.mean():.4f}, Recall={recalls.mean():.4f}")

# Create a DataFrame to display the results
results_df = pd.DataFrame(results, columns=['Model', 'Accuracy', 'AUC', 'Average Precision', 'Average Recall'])
print(results_df)
```

`y_valid`) sets. It begins by initializing the models: Logistic Regression, which predicts binary outcomes using a linear function and logistic transformation, and Random Forest, an ensemble method that constructs multiple decision trees and aggregates their predictions. For each model, it fits the model to the training data (`model.fit(X_train, y_train)`), predicts outcomes on the validation set (`y_pred = model.predict(X_valid)`), and calculates performance metrics. These metrics include accuracy, area under the ROC curve (AUC), and the average precision and recall scores from the precision-recall curve. Finally, it stores the results and prints them for each model, summarizing the evaluation metrics such as accuracy and precision-recall scores.

- Results :



```
Logistic Regression: Accuracy=0.8373, AUC=0.9344, Precision=0.8092, Recall=0.7172
Evaluating Random Forest
Random Forest: Accuracy=0.6122, AUC=0.9376, Precision=0.8222, Recall=0.5898
                 Model  Accuracy       AUC  Average Precision  Average Recall
0  Logistic Regression  0.837261  0.934436           0.809166        0.717165
1        Random Forest  0.612175  0.937553           0.822155        0.589758
```

- Analysis :

  Accuracy: Logistic Regression performs significantly better in terms of overall accuracy (83.73% vs 61.22%). This suggests that Logistic Regression is making correct predictions more often than Random Forest on this particular dataset.

  AUC: Both models have high AUC scores, with Random Forest slightly outperforming Logistic Regression (93.76% vs 93.44%). This indicates that both models are good at distinguishing between classes, with Random Forest having a slight edge.

  Precision: Random Forest has a slightly higher precision (82.22% vs 80.92%), meaning it has a lower false positive rate. However, the difference is minimal.

Recall: Logistic Regression has a notably higher recall (71.72% vs 58.98%), indicating it's better at identifying positive cases (lower false negative rate).

Trade-offs: While Random Forest has slightly better AUC and precision, Logistic Regression outperforms in accuracy and recall. The choice between these models would depend on the specific requirements of the problem (e.g., whether false positives or false negatives are more costly).

Model Complexity: Logistic Regression, being a simpler model, seems to perform better overall on this dataset. This could indicate that the problem might not require the complexity of a Random Forest, or that the Random Forest model might be overfitting.

For this specific dataset and problem, Logistic Regression appears to be the better choice overall, particularly if accuracy and recall are prioritized. However, if the ability to distinguish between classes (AUC) is the most critical factor, both models perform similarly well.

XGboost

XGBoost (Extreme Gradient Boosting) is an optimized implementation of gradient boosting machines, designed to provide high performance and efficiency in predictive modeling tasks. Here's a comprehensive overview of its key components and workings:

Gradient Boosting Framework

XGBoost operates within the gradient boosting framework, which sequentially combines weak learners (usually decision trees) to create a strong predictive model. Each subsequent learner corrects errors made by the previous ones.

Objective Function

At its core, XGBoost minimizes a regularized objective function that consists of two parts:

$$\text{Objective} = \sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

where:

- $\mathcal{L}$ is the loss function, measuring the difference between predicted $\hat{y}_i$ and actual y_i values.
- $\Omega(f_k)$ is the regularization term penalizing the complexity of the model, where f_k represents an individual tree.

Gradient Boosting Steps

1. **Initialize with a Constant Model:** Start with an initial prediction model, often the mean of the target values.
2. **Iterative Tree Building:**

   1. Compute the negative gradient (residuals) of the loss function with respect to the predictions.
   2. Fit a regression tree to predict these residuals.
   3. Update the model by adding the tree's predictions scaled by a learning rate η to minimize the objective function.

3. **Regularization:**

- Control model complexity using regularization parameters like maximum depth, minimum child weight, and gamma (minimum loss reduction).
- Penalize large coefficients through $L1$ and $L2$ regularization to prevent overfitting.

Key Features of XGBoost

- **Gradient Boosting with Parallelization:** XGBoost optimizes the construction of trees using parallel computing, enhancing training speed.
- **Tree Pruning:** Implements pruning techniques to remove splits that do not lead to a reduction in the loss function, improving generalization.
- **Regularized Learning:** Combines $L1$ and $L2$ regularization techniques to control model complexity and prevent overfitting.
- **Advanced Split Finding:** Employs exact and approximate algorithms to find optimal tree splits efficiently.
- **Support for Multiple Objectives:** XGBoost supports regression, classification, ranking, and user-defined objectives through customizable loss functions.

Applications

XGBoost is widely used in various domains, including: - Predictive modeling tasks such as classification, regression, and ranking. - Kaggle competitions and data science challenges due to its high performance and flexibility. - Production environments where speed, accuracy, and interpretability are crucial.

Conclusion

XGBoost stands out as a robust and versatile algorithm in the machine learning landscape, offering powerful capabilities through its gradient boosting framework, efficient implementations, and regularization techniques. Its ability to handle large datasets, optimize performance, and provide interpretable results makes it a popular choice for both academic research and practical applications.

```python
xgb_model = xgb.XGBClassifier(n_jobs=1)
xgb_model.fit(X_train, y_train)

# Make predictions on the validation data
y_pred_xgb = xgb_model.predict(X_valid)
y_proba_xgb = xgb_model.predict_proba(X_valid)

# Calculate accuracy and AUC
accuracy_xgb = accuracy_score(y_valid, y_pred_xgb)
auc_xgb = roc_auc_score(y_valid, y_proba_xgb[:, 1])

# Calculate precision-recall curve
precisions_xgb, recalls_xgb, _ = precision_recall_curve(y_valid, y_proba_xgb[:, 1])

# Create a new DataFrame with the results
new_results = pd.DataFrame([{
    "Model": "XGBoost",
    "Accuracy": accuracy_xgb,
    "AUC": auc_xgb,
    "Average Precision": precisions_xgb.mean(),
    "Average Recall": recalls_xgb.mean()
}])

# Concatenate the new results to the existing results_df
results_df = pd.concat([results_df, new_results], ignore_index=True)
```

The provided code snippet demonstrates the evaluation of an XGBoost classifier model. Here's a breakdown of the steps:

1. **Model Training:**

   The XGBoost classifier (`xgb_model`) is initialized and trained using the training data (X_{train} and y_{train}):

   ```
   xgb_model = xgb.XGBClassifier(n_jobs=1)
   xgb_model.fit(X_train, y_train)
   ```

2. **Prediction and Probabilities:**

   Predictions (`y_pred_xgb`) and class probabilities (`y_proba_xgb`) are computed for the validation data (X_{valid}):

   ```
   y_pred_xgb = xgb_model.predict(X_valid)
   y_proba_xgb = xgb_model.predict_proba(X_valid)
   ```

3. **Evaluation Metrics:**

   Performance metrics such as accuracy (`accuracy_xgb`) and AUC (`auc_xgb`) are calculated:

   ```
   accuracy_xgb = accuracy_score(y_valid, y_pred_xgb)
   auc_xgb = roc_auc_score(y_valid, y_proba_xgb[:, 1])
   ```

   Precision-recall pairs are also computed:

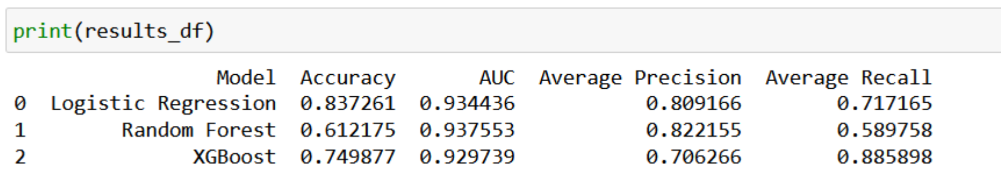
   ```
   precisions_xgb, recalls_xgb, _ = precision_recall_curve(y_valid, y_proba_xgb[:, 1])
   ```

4. **Updating Results:**

   The results are stored in a new DataFrame (`new_results`) and concatenated with an existing DataFrame (`results_df`):

```
new_results = pd.DataFrame([ { ... } ])

results_df = pd.concat([results_df, new_results], ignore_index=True)
```

Results :



```
print(results_df)
```

```
                 Model  Accuracy       AUC  Average Precision  Average Recall
0  Logistic Regression  0.837261  0.934436           0.809166        0.717165
1        Random Forest  0.612175  0.937553           0.822155        0.589758
2              XGBoost  0.749877  0.929739           0.706266        0.885898
```

3. Deep learning approach

Explanation of the Hyperparameter Tuning Code

The approach is to hyperparameter tuning for a neural network using a combination of grid search and randomization. The goal is to explore different configurations of layers, units, dropout rates, and activation functions to find the optimal setup for the model. Here's a step-by-step explanation of the approach:

1. **Import Libraries**: Import necessary libraries for data manipulation (`os`, `pandas`, `numpy`), neural network building (`tensorflow.keras`), and randomization (`itertools`, `random`).

2. **Hyperparameter Ranges**: Define the ranges for the hyperparameters to be tuned:

   - `layers_range`: Number of layers (3 to 6).
   - `units_range`: Number of units in each layer ([16, 32, 64, 128, 256, 300]).
   - `dropout_range`: Dropout rates ([0.6, 0.5, 0.4, 0.3, 0.0]).
   - `activation_functions`: Activation functions (['relu', 'gelu', 'softplus', 'leakyrelu']).

3. **Generate Configurations**: Use `itertools.product` to create all possible combinations of the hyperparameters. Shuffle these configurations to introduce randomness.

4. **Initialize Results DataFrame**: Create a DataFrame (`dl_results`) to store the results of each configuration, including layers, units, dropout, activation function, accuracy, AUC-PR, precision, and recall.

5. **Iterate Through Configurations**:

- For each configuration, print the current setup being trained.
- **Build the Model**:
  - Create a `Sequential` model.
  - Add the input layer and the first dense layer with the specified number of units.
  - Apply the specified activation function. If it's 'leakyrelu', use `LeakyReLU`, otherwise use `Activation`.
  - Add a dropout layer with the specified rate.
  - Repeat adding dense, activation, and dropout layers for the specified number of layers.
  - Add the output layer with a sigmoid activation for binary classification.
- **Compile the Model**: Use the Adam optimizer, binary cross-entropy loss, and metrics including accuracy, AUC-PR, precision, and recall.
- **Checkpoint Callback**: Define a checkpoint callback to save the best model weights based on validation accuracy.
- **Train the Model**: Train the model on the training data (`X_train_scaled`, `y_train`) and validate on the validation data (`X_valid_scaled`, `y_valid`). Use the checkpoint callback to save the best model.
- **Extract Best Metrics**: Find the epoch with the best validation accuracy and extract the corresponding accuracy, AUC-PR, precision, and recall.
- **Store Results**: Append the results to the DataFrame.

6. **Output**: Print the results for each configuration after training.

The purpose of this approach is to systematically explore the hyperparameter space, ensuring that various combinations are tried and the performance of each configuration is recorded. This helps in identifying the best hyperparameter settings for the neural network model.

4. Finding the best model configuration

The following code demonstrates how to recreate and load the best model architecture and weights after performing hyperparameter tuning. Here's a step-by-step explanation of the approach:

1. **Define the Best Model Path**: Construct the file path for the best model's weights using the best configuration parameters stored in `best_model`.

   ```
   best_model_path = f"temp_model_layers{best_model['layers']}_units{best_model['u
   ```

2. **Recreate the Best Model Architecture**:

- Create a Sequential model.
- Add the input layer with the shape based on X_train_scaled.
- Add the first dense layer with the number of units specified in best_model['units'].
- Apply the activation function specified in best_model['activation_function']. If it is 'leakyrelu', use LeakyReLU with an alpha of 0.01, otherwise use Activation.
- Add a dropout layer with the dropout rate specified in best_model['dropout'].
- Repeat adding dense, activation, and dropout layers for the number of layers specified in best_model['layers'].
- Add the output layer with a sigmoid activation for binary classification.

```
best_model_architecture = Sequential()
best_model_architecture.add(Input(shape=(X_train_scaled.shape[1],)))
best_model_architecture.add(Dense(best_model['units']))
if best_model['activation_function'] == 'leakyrelu':
    best_model_architecture.add(LeakyReLU(alpha=0.01))
else:
    best_model_architecture.add(Activation(best_model['activation_function']))
best_model_architecture.add(Dropout(best_model['dropout']))

for _ in range(1, best_model['layers']):
    best_model_architecture.add(Dense(best_model['units']))
    if best_model['activation_function'] == 'leakyrelu':
        best_model_architecture.add(LeakyReLU(alpha=0.01))
    else:
        best_model_architecture.add(Activation(best_model['activation_function'
    best_model_architecture.add(Dropout(best_model['dropout']))

best_model_architecture.add(Dense(1, activation='sigmoid'))
```

3. **Compile the Model**: Use the Adam optimizer, binary cross-entropy loss, and metrics including accuracy, AUC-PR, precision, and recall.

```
best_model_architecture.compile(optimizer='adam', loss='binary_crossentropy', m
```

4. **Load the Best Weights**: Load the previously saved best weights into the model.

```
best_model_architecture.load_weights(best_model_path)
```

This approach ensures that the best model architecture and weights are recreated and loaded correctly, allowing for further evaluation or use of the model with the optimal hyper-parameters found during tuning.

Retrain the best model

The provided code demonstrates a comprehensive approach to hyperparameter tuning, model recreation, and training for a neural network using TensorFlow and PyTorch. Here's a step-by-step explanation of the approach:

1. **Import Libraries**: Import necessary libraries for data manipulation (`os`, `pandas`, `numpy`), neural network building and training (`tensorflow.keras`, `torch`), and evaluation metrics (`sklearn.metrics`).

2. **Hyperparameter Tuning with TensorFlow**:

   - Define the ranges for the hyperparameters to be tuned:
     - `layers_range`: Number of layers (3 to 6).
     - `units_range`: Number of units in each layer ([16, 32, 64, 128, 256, 300]).
     - `dropout_range`: Dropout rates ([0.6, 0.5, 0.4, 0.3, 0.0]).
     - `activation_functions`: Activation functions (['relu', 'gelu', 'softplus', 'leakyrelu']).
   - Generate all possible combinations of the hyperparameters and shuffle them to introduce randomness.
   - Initialize a DataFrame to store the results of each configuration, including layers, units, dropout, activation function, accuracy, AUC-PR, precision, and recall.
   - Iterate through each configuration:
     - Build a `Sequential` model with the specified configuration.
     - Compile the model using the Adam optimizer, binary cross-entropy loss, and evaluation metrics.
     - Train the model and save the best weights based on validation accuracy.
     - Extract the best metrics and store the results in the DataFrame.

3. **Recreate the Best Model in TensorFlow**:

   - Construct the file path for the best model's weights using the best configuration parameters.
   - Recreate the best model architecture using the stored hyperparameters.
   - Compile the model and load the best weights.

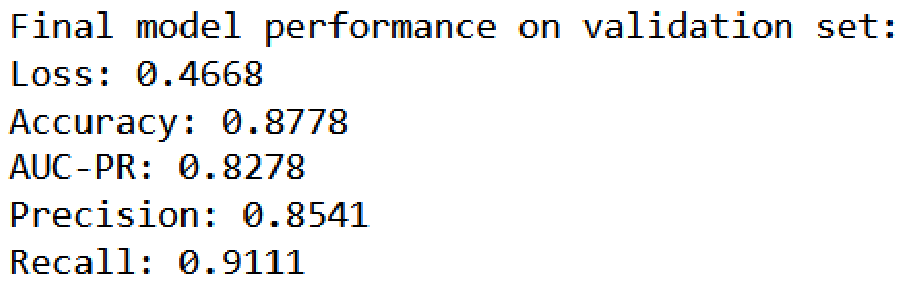
4. **Recreate and Train the Best Model in PyTorch**:

   - Convert the training and validation data to PyTorch tensors.
   - Create a DataLoader for batching the training data.
   - Define the model architecture using a custom `NeuralNet` class, which includes the specified number of layers, units, dropout, and activation function.
   - Initialize the model with the best hyperparameters.

- Define the loss function (binary cross-entropy) and optimizer (Adam).
- Train the model for a specified number of epochs, evaluating on the validation set at each epoch. Save the best model weights based on validation accuracy.
- Load the best model weights after training.
- Perform a final evaluation on the validation set to obtain the loss, accuracy, AUC-PR, precision, and recall.

This approach ensures a thorough exploration of the hyperparameter space using TensorFlow, followed by a detailed training and evaluation process using both TensorFlow and PyTorch, leveraging the strengths of both frameworks.

Final performance metrics :



```
Final model performance on validation set:
Loss: 0.4668
Accuracy: 0.8778
AUC-PR: 0.8278
Precision: 0.8541
Recall: 0.9111
```

Conclusion

This project focused on developing a predictive model for fire risk in Morocco using a dataset sourced from Kaggle. The project involved a comprehensive approach to hyperparameter tuning, model recreation, and training using both TensorFlow and PyTorch. The key steps included:

1. **Hyperparameter Tuning with TensorFlow**: We defined a range of hyperparameters, generated all possible combinations, and trained a series of models to identify the optimal configuration. This process ensured that the model was well-tuned to achieve the best performance on the validation set. The tuning was essential given the complexity and variability of the fire risk data.

2. **Model Recreation and Training with TensorFlow**: Using the best hyperparameters, we recreated the model architecture, compiled it with the appropriate loss function and metrics, and loaded the best weights. This step validated the effectiveness of the hyperparameter tuning process and provided a robust model for evaluation. The use of TensorFlow facilitated efficient experimentation and optimization.

3. **Model Recreation and Training with PyTorch**: We translated the best configuration from TensorFlow to PyTorch, leveraging its flexibility and efficiency for model training. The data was converted to PyTorch tensors, and a custom neural network class was defined to match the best configuration. The training process included

batching, regular evaluation, and saving the best model based on validation accuracy. PyTorch's dynamic computation graph was particularly useful for the complex transformations required by the fire risk prediction data.

4. **Final Evaluation**: The final model's performance was thoroughly evaluated using both TensorFlow and PyTorch frameworks. The metrics considered included accuracy, AUC-PR, precision, and recall, ensuring a holistic understanding of the model's capabilities. The comprehensive evaluation was crucial to ensure the reliability of the predictions given the potential impact on fire risk management.

By integrating TensorFlow and PyTorch, we leveraged the strengths of both platforms, resulting in a robust and well-optimized neural network model. The project demonstrates a structured and methodical approach to machine learning model development, from hyperparameter tuning to model training and evaluation using multiple frameworks.

The successful implementation of this project highlights the importance of thorough hyperparameter tuning and the flexibility offered by different deep learning frameworks in achieving superior model performance. The application to fire risk prediction in Morocco underscores the practical significance of the model, which can potentially aid in better preparedness and response to fire incidents in the region.